Materialized views for P2P XML warehousing

Ioana Manolescu Spyros Zoupanos

INRIA Saclay–Île-de-France, 4 rue Jacques Monod, 91893 Orsay Cedex, France

firstname.lastname@inria.fr

Abstract

Nous abordons la gestion efficace de documents XML dans un réseau pair à pair structuré, basé sur une table de hachage distribuée. Nous présentons une approche permettant d'exploiter des vues matérialisées déployées indépendamment sur le DHT par les pairs. Ces vues sont utilisées pour répondre à des requêtes de motifs d'arbes XML. Nous proposons des algorithmes pour la maintenance distribuée de vues et pour la réécriture des requêtes en utilisant desvues. Nous présentons les différentes stratégies d'indexation et de recherche de vues sur le réseau DHT et nous comparons leur performance à travers des expériences sur notre plate-forme.

Keywords XML, query rewriting, materialized views, DHT.

1 Introduction

Large-scale organizations need to produce, deploy and exploit large volumes of data, and in particular structured XML data. We are currently involved in the WebContent R&D project (http://www.webcontent.fr), including partners such as EADS, the European defense company, and CEA, the French nuclear energy producer. The project focuses on giving these companies efficient tools for gathering, enriching and exploiting (e.g. by semantic and linguistic analysis) structured documents from the Web (RSS feeds, crawled pages etc.) or produced by the companies, within a *structured content warehouse*, used for market analysis or Web intelligence gathering. Information comes from many sites, depending on where crawlers run, or where the analyst producing a company report resides, and must be efficiently exploitable by the other sites. The sites producing information may change over time, as new ones may join and others may leave; such changes must be handled transparently to the users. However, the network dynamics is moderate, i.e., sites do not experience frequent disconnects.

Within WebContent, we have devised a platform called **ViP2P**, standing for *Vi*ews in *P*eer to *P*eer, which enables efficient distributed data management based on a DHT (*d*ynamic *h*ash *t*able [10]) and materialized XML views. ViP2P can be seen as a tool for *redistributing restructured data where it is needed*. Any ViP2P site (or peer) may establish some materialized views, which reflect data published anywhere in the network, that the peer is interested in. A more likely scenario is that several peers which are physically close (e.g. machines in the same company site) share the burden of storing some views which may be interesting to all of them. All view definitions are then indexed in the DHT, so that any peer may learn about them. A query posed on any peer is re-written using the existing views. In this work, we focus on the problem of finding equivalent query rewritings based on the views in the DHT, as well as on building and advertising the views.

This paper makes the following original contributions. (1) We describe the VIP2P architecture for managing and exploiting materialized XML views based on a DHT. (2) We consider the problem of tree pattern query rewriting problem based on multiple views. The sets of rewritings identified by our and similar works [4, 9, 18] partially overlap; we prove an interesting bound on the maximal rewriting size, making it polynomial in the number of views, and we study several corresponding rewriting algorithms.

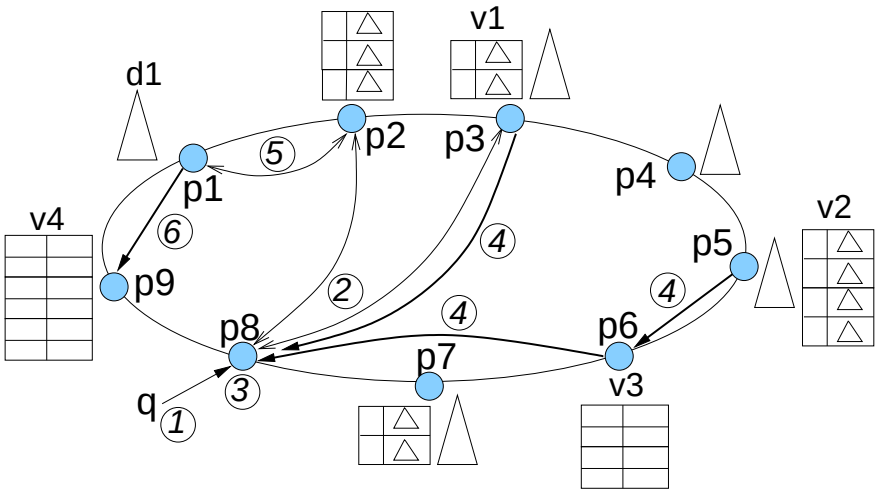

Figure 1: Architecture overview.

(3) We study several strategies for indexing materialized view definitions on a DHT, and compare their usefulness. (4) We demonstrate via experiments in a fully implemented platform the scalability of our platform.

The architecture we envision is depicted in Figure 1. Network peers labeled p_1 to p_9 store documents, shown as triangles, and/or views, shown as tables. Such tables attributes may be of type xml (whose values are serialized XML subtrees), in the style of the native XML data type in SQL/XML 2003. Such attributes are shown as triangles inside tuples. We designate a document d or view v at peer p by the notation $d@p$, respectively, $v@p$.

Each view is defined by a tree pattern, and this pattern (not the view extent) is indexed in the DHT. Query processing can be traced following the numbered arrows in the Figure. Assume query q is asked at peer p_8 (step 1). Then, p_8 will perform a DHT look-up to find which view definitions may be useful to rewrite the query. For instance, p_2 and p_3 may return to p_8 relevant view definitions (step 2). Peer p_8 will then run a view-based query rewriting algorithm, trying to reformulate q based on these definitions (step 3). A query rewriting is a logical algebraic plan based on some views, in our example, $v_1@p_3$, $v_2@p_5$, and $v_3@p_6$. After picking a rewriting, p_8 transforms it into a distributed physical plan, which runs in a distributed fashion (step 4, thick arrows denote data transfer). In our example, v_2 is sent to p_6 which joins it with v_3 and sends the result to p_8. At p_8 it joins with v_1 which is sent from p_3.

Each ViP2P view v is complete, i.e. it includes $v(d)$ for any document d in the network (modulo some update propagation time). To obtain such views, whenever a new document, say $d_1@p_1$ in Figure 1, is published, the publishing peer performs another type of lookup (step 5) to determine (possibly a superset of) the view definitions to which the new document may contribute tuples. In the Figure 1, such definitions are returned by p_2, and p_1 finds out that d_1 contributes some tuples to the view $v_4@p_9$. The tuples are sent to p_9 (step 6), which adds them to the view extent.

This article is organized as follows. Our pattern language is discussed in Section 2. Section 3 presents the rewriting problem and its complexity, and Section 4 studies several rewriting algorithms. How views are materialized, indexed in the P2P network, and looked up is discussed in Section 5. We present our experiments in Section 6, discuss related works in Section 7, then conclude.

2 Patterns

We will rely on a tree pattern dialect $\mathcal{P}$, defined as follows.

(1) Pattern nodes can correspond to *XML internal nodes* (elements or attributes), or to *leaves* (words in text occurring inside XML elements, or in attribute values). For presentation purposes, we do not distinguish between elements and attributes. We extend the XPath descendant axis to consider that words are children of their closest element or attribute ancestors. Each internal pattern node carries a label from a tag alphabet $A_l = \{a, b, c, \ldots\}$. Each leaf node carries a label from a word alphabet $A_w = \{\underline{a}, \underline{b}, \underline{c} \ldots\}$.
(2) Pattern edges correspond to parent-child or ancestor-descendant relationships between nodes.
(3) Each pattern node may be annotated with some *stored attributes*, describing some information items that the pattern stores out of each XML node matching the pattern node. The $cont$ annotation indicates that the full (serialized) image each matching XML tree node is stored. The id annotation indicates that a node identifier, which uniquely identifies the node (and the document it belongs to). Moreover, we assume *structural* IDs, i.e. such that one may decide, by comparing the identifiers of two nodes n_1 and n_2, whether n_1 is an ancestor/parent of n_2 or not. Many variants of structural identifiers exist, e.g., [2, 12, 20], some of which provide further information, e.g. allow identifying the least common ancestor of two nodes etc. *For the purpose of this work, we only require that parent-child and ancestor-descendant relationships can be determined from the node IDs.* Finally, the val labels stands for the node's text value, obtained by concatenating all its text descendants in document order.
(5) Each node may be annotated with a predicate of the form $[val = \underline{c}]$ where $\underline{c} \in A_w$, restricting the XML nodes which match the pattern node, to those satisfying the predicate.
Notations and syntax simplification We say a pattern node *has* an id, respectively val, $cont$, or value predicate, if the node is decorated with such an index.

For simplification, in the sequel, we only consider ancestor-descendant node relationships. All the results we present hold in the presence of both ancestor-descendant and parent-child relationships; the distinctions to be made are quite well-known by now, e.g. when computing pattern embeddings [3], or structural joins [2].

We introduce a simple text syntax for patterns. We denote nodes by their A_l or A_w label. The possible id, val and $cont$ labels, and predicates over val, are shown as indices to the node. For instance, $a_{id\,cont}$ is a pattern storing the structural IDs and the content of all a elements. We use parenthesis to show the nesting of children inside parents, and commas to separate the children of the same pattern node among themselves. For instance, $a(b(c_{id}))$ stores the IDs of elements found on some path matching $//a//b//c$. The pattern $a_{[val=5]}(\underline{b}, c_{id})$ stores the identifiers of all c elements having an a ancestor of value 5, and whose serialized XML subtree contains the word $\underline{b}$.
Pattern semantics Let p be a pattern and d be an XML document. As customary, an embedding $\phi : p \to d$ of p in d is a function associating d nodes to p nodes, preserving node labels and ancestor-descendant relationships [3]. The result of evaluating p on d, denoted $p(d)$, is the list of tuples obtained by lining together in a tuple, all IDs and/or values and/or serialized content, for each embedding of p in d. Assuming a total order over the nodes of p (top-down, left-to-right traversal), the tuple order in $p(d)$ is dictated by the lexicographic order over the d nodes which are targets of the embeddings. For a document set $\mathcal{D}$, the semantics of p over $\mathcal{D}$ is defined as the concatenation (in the order of the document IDs) of all $p(d)$, $d \in \mathcal{D}$.

We use $a.id$ (respectively, $a.val$, $a.cont$) to denote the corresponding attribute in $p(\mathcal{D})$.

We say two patterns p_1, p_2 are *equivalent*, denoted $p_1 \equiv p_2$, if for any database $\mathcal{D}$, $p_1(\mathcal{D}) = p_2(\mathcal{D})$. We establish containment and equivalence of $\mathcal{P}$ patterns in time polynomial in the size of the patterns [3].

3 Algebraic rewritings using patterns

Given a query $q \in \mathcal{P}$ and a set $\mathcal{V}$ of views, we are interested in the rewritings of q, based on $\mathcal{V}$. As explained in Section 2, the semantics of both queries and views are *relations*, therefore, we investigate rewritings which combine views by means of a relational algebra, specified in Section 3.1. Based on this, Section 3.2 formally states the rewriting problem, while Section 3.3 show that its complexity is polynomial in the number of views.

3.1 Algebra

The algebra we consider consists of the following operators:
(1) $scan(v)$ (or v, in short), where $v \in \mathcal{V}$ is a view.
(2) The n-ary cartesian product operator $\times$, projection (denoted π_{cols}), duplicate elimination (denoted π^o), and sort (denoted s_{cols}).
(3) Selection, denoted σ_{pred}. Here, $pred$ is a conjunction of predicates of the form $i \odot \underline{c}$ or $i \odot j$, where i, j are attribute names, $\underline{c} \in A_w$, and $\odot \in \{=, \prec\}$ is a binary operator. We use $\prec$ to designate the "is ancestor of" predicate. Thus, assuming the attributes named i and j contain IDs, $\sigma_{i \prec j}(op)$ returns those op tuples where the identifier in attribute i corresponds to an ancestor of the node whose identifier is in attribute j.

Note that the presence of $\times$ and σ allows, in particular, ID equality joins $\bowtie_{=}$, as well as structural joins [2], denoted $\bowtie_{\prec}$.
(4) A navigation operator, designated $nav_{i,np}$, which takes as input one algebraic expression. The attribute i in the input must correspond to a $cont$ attribute, and np is a pattern whose nodes must not have ids. Let t be an input tuple to the nav, and $np(t.i)$ denote the result of evaluating the pattern np on the XML fragment stored in $t.i$ (as defined in Section 2). Then, $nav_{i,np}$ will output the tuples $t \times np(t.i)$, i.e., obtained by successively appending to t each of the tuples in $np(t.i)$. If $np(t.i)$ is empty, $nav_{i,np}$ acts like a selection, erasing t. The reason why np nodes must not have ids is that it is generally not possible to determine the ID of a node, from an XML fragment (not the whole document) to which the node belongs.

As an example, let v be the view $a_{id\ cont}$. The expression $e = nav_{a.cont,b(c_{cont},d_{cont})}(v)$ returns tuples with 4 attributes: a identifiers, a contents, and contents of c and d descendants of a, having a common b ancestor below a.

For convenience, we extend the notation to allow several patterns to be applied on the same $cont$ attribute by a single nav operator. Thus, $nav_{i,np_1,np_2}(op) = nav_{i,np_2}(nav_{i,np_1}(op))$.

3.2 Problem statement

Equivalent rewritings Let q be a $\mathcal{P}$ query, and $e(v_1, v_2, \ldots, v_k)$ be an algebraic expression over the views in $\mathcal{V}$. We say $e(\mathcal{V})$ is an equivalent rewriting of q if and only if, for any database $\mathcal{D}$, $e(v_1(\mathcal{D}), v_2(\mathcal{D}), \ldots, v_k(\mathcal{D})) = q(\mathcal{D})$.

As an example, the expression e from the last example above is an equivalent rewriting for the query $q = a_{id,cont}(b(c_{cont}, d_{cont}))$.

Problem statement (first attempt) We may at this point specify our problem as: given q and $\mathcal{V}$, find all equivalent rewritings of q using the views $\mathcal{V}$. Here and in the sequel, we assume the views and the query have been minimized as in [3] (a difference to be made for our patterns with attributes is: no node having id, $cont$ or val can be removed by minimization).

However, this problem definition leads to an artificially large space of solutions, since two algebraic expressions may differ in their view join orders, selection and projection positions etc., all the while corresponding to the same rewriting. For instance, let $q = a_{id}(b, c, d)$ and $v_b = a_{id}(b)$, $v_c = a_{id}(c)$, $v_d = a_{id}(d)$. Twelve syntactically different join expressions over v_b, v_c and v_d are equivalent rewritings of q. We are not interested in exploring these alternatives during rewriting, as this exploration pertains to the subsequent algebraic optimization step. To that effect, we introduce the notion of *canonical algebraic expressions*. An algebraic expression e is said to be canonical if it has one of the following forms:

- form 1: $scan(v)$ or $nav(scan(v))$
- form 2: $\times(\alpha_1, \ldots, \alpha_k)$, where each α_i is of form 1
- form 3: $\sigma_{pred}(\beta)$, where β is of form 1 or 2
- form 4: $s_{cols}(\gamma)$, where γ is of form 1, 2 or 3
- form 5: $\pi_{cols}(\delta)$, where δ is of form 1, 2, 3 or 4

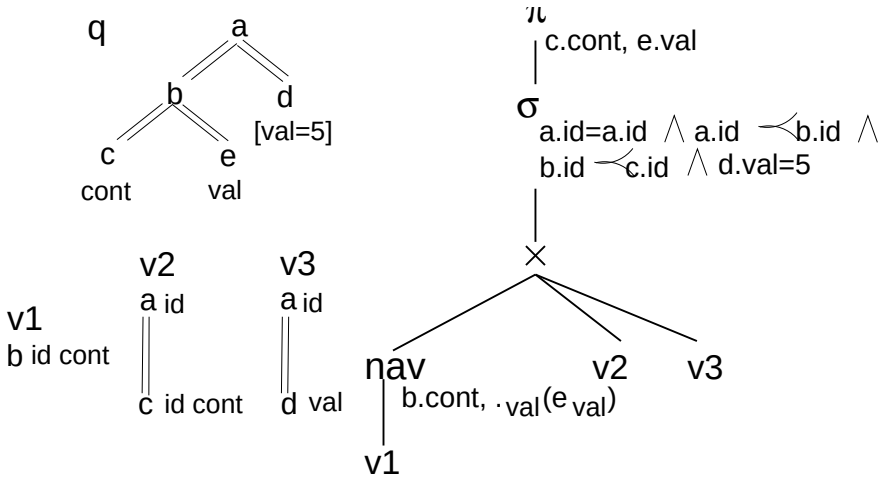

Figure 2: Sample query, views, and rewriting.

- form 6: $\pi^0(\epsilon)$, where ϵ is of form 1, 2, 3, 4 or 5.

Intuitively, the operators in canonical expressions are (1) consolidated - there will be at most one of each of the following operators: cartesian product, selection (possibly on a conjunction of predicates), sort, projection, and duplicate elimination and (2) applied in a specific order ($scan$, then nav, then $\times$, σ, s, π and π^0 respectively).

Any algebraic expression can be brought to a canonical form. We say e is a *canonical rewriting* of q if e is a rewriting of q, and e is a canonical expression.

Minimal rewritings Certain canonical rewritings exhibit some redundancy, as illustrated by the query $q = a_{id}$ and identical views $v = v' = a_{id}$. Then, $e_1 = v$ and $e_2 = v'$ are canonical rewritings, but so is $e_3 = \pi_{id}(\sigma_{id=id}(v \times v'))$. Intuitively, we are interested in finding e_1 and e_2, but not e_3. More formally, let e be an algebraic expression. We say e is *minimal* if and only if all the expressions obtained by removing a view from e are not equivalent to e. Several minimal canonical rewritings can be obtained from a non-minimal one, as shown in the last example above.

View pruning If v appears in a rewriting of q, then there exists an embedding $\phi : v \rightarrow q$, such that:

1. ϕ preserves node names
2. if n is a parent of m in v, $\phi(n)$ is an ancestor of $\phi(m)$
3. if m has a value predicate $[val = c_1]$ in q and $\phi(n) = m$, for some v node m, then m must not have a value predicate $[val = c_2]$, if $c_1 \neq c_2$.

A similar observation has been made in [18] (excluding item 3 above). This observation allows pruning down the set of views $\mathcal{V}$ to a subset $\mathcal{U}$ of views which can be embedded in q, while being guaranteed not to lose any rewritings by doing so.

View expansion An important refinement of our problem is needed. From the $cont$ attribute of a view node, one may extract the value of this node's val attribute (e.g. by the XPath query $./text()$), as well as information about its descendants, since they appear in the $cont$ XML subtree. For instance, in Figure 2, one can expand v_1 by navigating within $b.cont$ to find its text value, and the text values of all its e descendants. This is represented by the algebraic plan $nav_{b.cont,\,.val(e_{val})}(v_1)$, where by a slight abuse of syntax, we denoted by $.val$ a pattern node matching only the root of the XML tree on which it is evaluated, and having a val attribute. The result can be seen as an expanded view $v_1' = b_{id,val,cont}(e_{val})$. The algebraic expression at right in Figure 2 builds on this plan, and is a canonical minimal rewriting for the query.

The need for partial expansions Observe that an expanded view does not necessarily add under the view node having $cont$, all the forest rooted at the corresponding query node. In the above example, expansion

added an e node but not a c node. Indeed, had we added the c node too, it would have been impossible to rewrite q. Let us see why. Assume expansion transforms v_1 into $v_1'' = b_{id,val,cont}(c_{cont}, e_{val})$. We may join v_1'' with v_2 enforcing that $a.id$ is an ancestor of $b.id$ and $b.id$ is an ancestor of $c.id$ (the latter from v_2). The resulting expression has two c nodes, descendants of b: the one from v_1'' has $cont$, while the other has id and $cont$. As a result, this expression contains a cartesian product of the $//b//c$ nodes with themselves, which was not required by the query. We cannot unify the two c nodes, as the one from the expanded view v_1'' does not have id. Thus, it is impossible to rewrite q based on v_1''; v_1' is necessary. This phenomenon is due to the fact that unlike XPath views used e.g. in [18], our views may store data from more than one nodes.

We consider the views in $\mathcal{U}$ have all been expanded into a set $\mathcal{W}$, and reason on $\mathcal{W}$ from now on.

Problem statement (final) Our problem can be now stated as follows: given a query q and a set of views $\mathcal{V}$, find all minimal canonical rewritings of q using views from the set $\mathcal{W}$, obtained by pruning, and then expanding, $\mathcal{V}$.

3.3 Complexity

Several aspects impact rewriting complexity.

View pruning is performed by evaluating each view on the query, considered as a data tree; if the result is non-empty, an embedding has been found, and the view is kept. Thus, the cost of obtaining the set $\mathcal{U}$ from $\mathcal{V}$ is $\Theta(|q| \times \Sigma_{v \in \mathcal{V}}|v|)$.

All views which survive pruning have at most as many nodes as the query (recall also that they are minimized from the start). Thus, for any $v \in \mathcal{U}$, $|v| < |q|$.

Expansion impact We consider the size of the set $\mathcal{W}$ obtained by expanding $\mathcal{U}$ views.

Let v be a view in $\mathcal{U}$, where a single node m has a $cont$. Assume an embedding ϕ maps m to the query node n. In principle, we should generate $2^{|n|} - 1$ copies of v (where $|n|$ is the size of the query tree rooted at n), each of which copies as a new child of m, a subtree of the n-rooted query tree. If a node in this subtree has an id, the id will be erased in the copy, since as said in Section 3.1, IDs cannot be found inside $cont$ attributes.

Three observations allow to reduce this set (see Figure 3):

1. Let n_1 be a descendant of n, and n_2 be a child of n_1. Let v' be an expansion of v, in which n_1 is copied as n_1', whereas n_2 is not copied. To build a rewriting based on v', we would need some other view v_x covering node n_2, and a predicate of the form $n_1'.id \prec n_2.id$ enforcing the appropriate relationship between the two nodes. However, n_1' lacks an ID, thus this predicate cannot be checked, and v' is useless. Thus, we only develop the expanded views such that: if a descendant n_1 of n is copied, so are its children, but also their children etc. - thus, the full subtree rooted at n_1 must be copied.

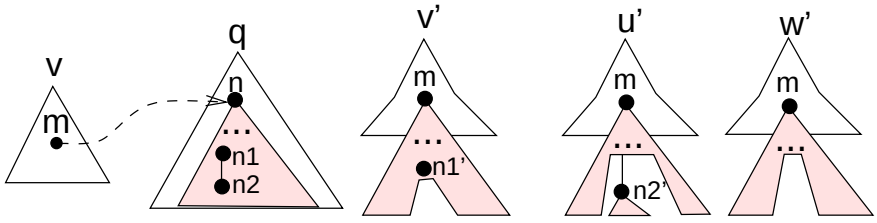

Figure 3: View expansion.

2. Again, let n_1 be a descendant of n, and n_2 a child of n_1. This time, we consider an expanded view u' where n_2 is copied as n_2', and n_1 is not copied. To build a rewriting based on u', we would need another view covering n_1, and a predicate of the form $n_1.id \prec n_2'.id$ enforcing the appropriate relationship among the two nodes. Again, this predicate cannot be checked since n_2' lacks an ID. Therefore, if we copy n_2, all its ancestors up to n must be copied.

3. In the case when m does not have an id, let n_1 be a descendant of the query node n, and w' be an expanded view such that n_1 is not copied in w. By a similar argument as above, a rewriting based on w' needs another view covering node n_1, and a predicate over $n_1.id$ ensuring that it is a descendant of m's copy in w'. Since the latter node does not have an ID in v nor w', no rewriting can use w'.

Observations 1. and 2. entail that we only need to enumerate the subsets of n children, and for each subset, build an expanded pattern which fully copies the subtrees rooted in those children. This reduces the number of generated expanded views from $2^{|n|}$ to $2^{f(n)}$ (where $f(n)$ is the fan-out of n), which is often smaller. Observation 3. further reduces it to 1 in the particular case where m does not have an ID.

More generally, let $f(q)$ be the maximum fan-out of a $cont$ node in q, $f(q) \leq |q|$. Let v_i be a view whose nodes $m_i^1, m_i^2, \ldots, m_i^{k_i}$ have $cont$, $k_i \leq |v_i| \leq |q|$, and ϕ_i an embedding from v_i to q such that $\phi_i(m_i^j) = n_j$, $1 \leq j \leq k_i$. The expansion of v_i produces $\Pi_{j=1,\ldots,k_i}(2^{f(n_j)})$ views, which is bounded by $\Pi_{j=1,\ldots,k_i} 2^{f(q)} \leq 2^{|q| \times f(q)}$.

Thus, $|\mathcal{W}| \leq |\mathcal{V}| \times 2^{|q| \times f(q)} \leq |\mathcal{V}| \times 2^{|q|^2}$.

Rewritings and covers Let e be a canonical rewriting based on some a subset V of $\mathcal{W}$. Clearly, the set of q nodes can be seen as covered by the union of the node sets of the view involved in e. Thus, from a rewriting, one can extract a *query cover* based on the view nodes.

Not any query cover leads to a rewriting. For instance, consider the views $v_1 = a_{id}(b)$ and $v_2 = c_{id}$, and the query $q_1 = a_{id}(b(c))$. In this case, the query requires the b node to be an ancestor of the c node, but since v_1 does not store identifiers for b, we are unable to enforce this constraint.

A cover may use a view several times, and in distinct positions. Consider, for instance, $q_2 = a(a_{id})$, and the views $v_3 = v_4 = a_{id}$; q_2 may be covered (and rewritten) by using: v_4 twice, or v_3 twice, or v_3 in the ancestor role and v_4 in the descendant role, or the opposite.

More generally, *to each set of pairs of the form (view from $\mathcal{W}$, embedding from the view to the query), where distinct pairs may use the same view with different embeddings, corresponds at most one rewriting*. We will describe an algorithm which builds this rewriting when possible in Section 4; the algorithm runs in quadratic time in the combined size of the views.

How many different embeddings exist from a view to the query? If all query nodes have different labels, then this also holds for each view, and at most one embedding exists. In general, let $\nu(q)$ be the maximum number of times a given node label appears in the query. Then, the view can be embedded in at most $\nu(q)^{|v|} \leq \nu(q)^{|q|}$ ways.

Rewriting size bound As we will show in Section 4, the maximum number of views involved in a minimal canonical rewriting is equal to the number of query nodes.

Putting it all together, the worst-case complexity for enumerating all minimal canonical rewritings is of the form $|q| \times (\Sigma_{v \in \mathcal{V}}|v|) + (\Sigma_{k=1,\ldots,|q|} C^k_{|\mathcal{W}| \times \nu(q)^{|q|}}) \times (\Sigma_{v \in \mathcal{W}}|v|)^2$. The first term accounts for pruning. The second is the cost of enumerating all subsets of (view from $\mathcal{W}$, embedding) pairs, of size at most $|q|$ (by a known formula, this is in $O((|\mathcal{W}| \times \nu(q)^{|q|})^{|q|}/|q|!)$), multiplied by the quadratic cost of building a rewriting out of such a set. More simply, the sum of the combinations can be put as $O(|\mathcal{W}|^{|q|})$ which translates to $O(|\mathcal{V}|^{|q|})$. The last factor $(\Sigma_{v \in \mathcal{W}}|v|)^2$ is less than $(|\mathcal{W}| \times |q|)^2$, thus in $O(|\mathcal{V}|)^2$. Thus, the total cost is in $O(|\mathcal{V}|)^{|q|+2}$. This is significantly less than the $O(2^{|\mathcal{V}|})$ which can be attained with a naïve set-cover approach.

4 Rewriting-based query answering

In this section, we describe how queries can be answered in our architecture, based on materialized tree pattern views. Section 4.1 discusses if and how an algebraic rewriting can be built out of a set of views. Section 4.2 discusses algorithms for enumerating all minimal canonical rewritings. Section 4.3 outlines the optimization and execution of our rewriting plans.

Algorithm 1: Views-to-rewriting (possibly partial)

Input : query q, views $v_1, \ldots, v_k \in \mathcal{W}$,
embeddings $\phi_i : v_i \rightarrow q$, $1 \leq i \leq k$
Output: partial rewriting of q using $v_1, \ldots, v_k$, if one exists

1 $e \leftarrow \times(v_1, v_2, \ldots, v_k)$
2 **foreach** *query node* n **do**
3 $\quad S(n) \leftarrow \{n_i \in v_i$ such that $\phi_i(n_i) = n$ and n_i has an $id\}$
4 $pred \leftarrow$ true
5 **foreach** *query node* n **do**
6 $\quad pred \leftarrow pred \wedge \bigwedge_{n_i, n_j \in S(n)}(n_i.id = n_j.id)$
7 $\quad$ **foreach** m *child of* n **do**
8 $\quad\quad pred \leftarrow pred \wedge \bigwedge_{n_i \in S(n), m_j \in S(m)}(n_i.id \prec m_j.id)$
9 $e \leftarrow \sigma_{pred}(e)$;
10 $d \leftarrow DAG(e)$; $d \leftarrow minimize(d)$
11 **if** d *is not a tree, or* d *cannot be embedded in* q **then**
12 $\quad$ fail
13 attemptFinalize(e, q)
14 return e

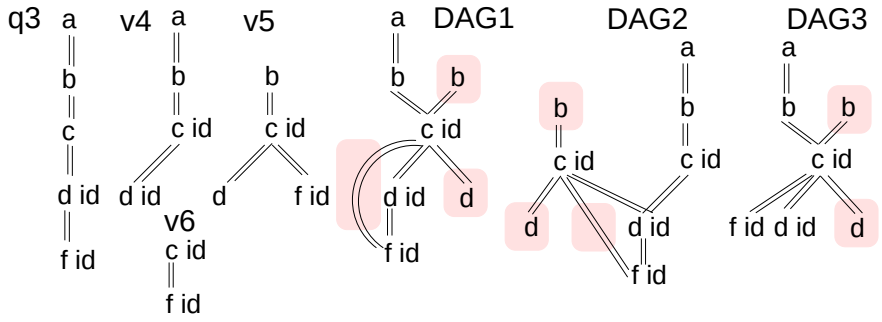

Figure 4: Sample query, views, and DAGs.

4.1 Building a rewriting out of a set of views

As we have seen, rewritings can be obtained out of some, but not all, covers of the query using the views. We present an algorithm (Algorithm 1) which, given the query, a subset of views in $\mathcal{W}$, and embeddings from each view into the query, builds a particular algebraic expression over all the views, or fails. In particular, if the views constitute a node cover, and if an expression is returned, it will be a canonical rewriting of q, using all the views. This rewriting is not guaranteed to be minimal; we will address minimality further on.

Algorithm 1 starts by building the cartesian product of the views. Then, it adds a selection on two kinds of predicates (lines 2-9). First, all the view nodes having an id, and embedded in the same query node, must be equal. Second, structural relationships between query nodes should be enforced over the corresponding view nodes. As an example, consider the query q_3 and the views v_4 and v_5 in Figure 4. They lead to the expression $e_1 = \sigma_{c.id=c.id \wedge c.id \prec d.id \wedge d.id \prec f.id}(v_4 \times v_5)$, where in the second predicate, c is from v_5 and d is from v_4. Observe that no predicate connects the b nodes, since they do not have ids.

At line 10 in Algorithm 1, we examine the resulting expression by means of a DAG representation, built as follows. Take all the v_i trees, and fuse all view nodes mapped to a same query node, into one node. (The result is guaranteed to be a DAG, since the embeddings ϕ_i map the v_i trees to q.) We then minimize

the DAG, by removing all redundant nodes, and redundant edges. A node is redundant if it has no id, val or $cont$ attribute, and is not the only one mapped to its corresponding query node. An edge is redundant if there exists a path in the DAG connecting the same nodes. In the example of Figure 4, DAG_1 is the DAG obtained from e_1; the nodes and edge on gray background are eliminated by minimization.

The function **attemptFinalize** attempts to build a q rewriting out of e. It first tests if an embedding from q to d can be found; if yes, this also implies that the views form a query cover. If so, it will attempt to add the value selection predicates from q that e does not have, a sort, a projection, and/or a duplicate elimination on top of e. The sort addition is a bit complex, since the order in which e produces results depends on the physical operators implementing it, and these operators are not known at this point. Thus, we check (a) if the existing view orders allow producing e outputs in the right order for q by *some* possible physical implementation of e, or (b) if not, if e projects sufficiently many IDs to enable sorting e's output as desired. If some of the desired operations cannot be added, attemptFinalize fails. In the example in Figure 4, the canonical rewriting found is $\pi_{d.id,f.id}(e_1)$.

Correctness Algorithm 1 is correct, i.e. if attemptFinalize produces an output and flags it as a complete rewriting (line 13), that is indeed canonical rewriting of q. This is guaranteed by the fact that e is equivalent to d (and the minimized d). If the minimized d is isomorphic to q, and the necessary selection, projection and sort operations could be applied on its equivalent expression e, then the result is an equivalent rewriting of q.

Completeness If a canonical rewriting based on a set of views and embeddings exists, Algorithm 1 produces it. This is because of the aggressive application of node predicates (lines 5-8), enforcing as many of the query-derived relationships between nodes as possible. Intuitively, omitting one of the predicates may lead to an undesired cartesian product in e. For instance, in Figure 4, if we omit the predicate $c.id = c.id$ from e_1, we obtain the DAG_2 from the same Figure, which, after minimization, is not a tree. If we omit the predicate $d.id \prec f.id$, we obtain the DAG_3 in Figure 4, which, after minimization, is a tree, but attemptFinalize cannot turn it into a rewriting, since f is not a descendant of d.

Complexity Algorithm 1 runs in quadratic time in the combined size of the views; the most expensive operation is the DAG minimization.

Bound on minimal rewriting size We now prove that a minimal canonical rewriting of q uses at most $|q|$ views.

We start by proving the following lemma: for any query node n, there must exist at least a view node m_i of a view v_i used in the rewriting, such that $\phi_i(m_i) = n$ and m_i has *at least as many attributes* as n. We distinguish possible cases by the number of attributes that n has:

- No attributes: the fact that at least one view v_i *must* have a node m_i mapping to n satisfies our claim.
- One attribute: the rewriting must provide it, so at least one of the m_i nodes mapped to n must have it.
- Two attributes: consider first the case when one attribute is an id. Either m_i is the only view node mapped to n (in which case m_i must also provide the other attribute), or there are several view nodes mapped to n. Among these, some may have no attributes at all, but those which do have attributes *must all* have ids, to enable the corresponding view join[1]. Among these joined views, having m_j nodes with ids such that $\phi_j(m_j) = n$, one at least must also provide the other attribute of n.
  Now consider the case when n has val and $cont$. Similarly, either m_i is the only view node mapped to n, thus it provides both; or, the rewriting joins several views by the equality of their nodes mapped to n. Among these nodes, some must have $cont$, and those who do, also have val due to expansion (Section 3.2).

[1]Otherwise, undesirable cartesian products (recall Partial Rewritings from Section 3.2) or, equivalently, non-tree DAGs (such as DAG_2 in Figure 4) may occur, and prevent rewriting.

Algorithm 2: Subset-enum

Input : query q, view set $\mathcal{V}$
Output: all minimal canonical rewritings of q based on $\mathcal{V}$

```
1 U ← prune(V, q); W ← ∪_{v∈V} expand(v)
2 R ← ∅
3 foreach qc = {v_1, v_2, ..., v_k} subset of W, |qc| ≤ |q| do
4     foreach tuple φ_1, φ_2, ... φ_k of embeddings from v_1, v_2, ..., v_k into q do
5         e ← views-to-rewriting(qc, φ_1, φ_2, ... φ_k) (use Algorithm 1)
6         if e is an equivalent rewriting then
7             add e to R
8 remove from R non-minimal rewritings
9 return R
```

- Finally, if n has id, val and $cont$, either m_i is the only view node mapped to n and it has these attributes, or several views are joined on ids of nodes mapped to n. The first one to have $cont$, also has val, due to expansion.

Let $v_1, \ldots, v_k$ be an ordering over the views in the rewriting. Based on the lemma, we define the *contribution* of v_i, denoted $C(v_i)$, as the set of query nodes n, such that (a) a node of v_i is mapped to n and has at least all the attributes of n, and (b) no view appearing *before* i in the rewriting satisfies this, i.e., for all $j < i$, either v_j does not have a node mapped to n, or that node does not have all the attributes of n.

It is easy to see that for two distinct views v_i, v_j, the sets $C(v_i)$ and $C(v_j)$ are disjoint.

A view v_i such that $C(v_i) = \emptyset$ is redundant. This is because any node relationship, or attribute, which v_i brings to the rewriting, can also be found using the previous views. Thus, for v_i not to be redundant, $|C(v_i)|$ must be at least 1.

Finally, the union of the $C(v_i)$ sets, for all views v_i, is the set of the query nodes. Thus, at most $|q|$ views participate to a minimal rewriting. □

For example, consider the rewriting of q_3 based on v_4 and v_5 discussed above. In this case, $C(v_4) = \{a, b, c, d\}$, and $C(v_5) = \{f\}$. This rewriting is minimal. Now assume that we also add the view v_6 from the Figure to the rewriting, before v_4 and v_5. Then, $C(v_6) = \{c, f\}$, $C(v_4) = \{a, b, d\}$, $C(v_5) = \emptyset$ and v_5 is redundant.

DAG vs. rewriting minimality Algorithm 1 minimizes the DAG d, not the rewriting e. Using the C sets, one can extract from a non-minimal rewriting e *some* minimal one, in time polynomial in $|q|$.

4.2 Rewriting algorithms

The first end-to-end rewriting algorithm we consider is called **Subset-Enumeration**, or **SE** in short (Algorithm 2). It iterates over all $\mathcal{W}$ subsets of size at most $|q|$, all embedding combinations from the views into q, and accumulates rewritings in the set R. A rewriting r is non-minimal if another rewriting $r' \in R$ uses a subset of r's views.

Algorithm SE does not specify a subset enumeration order; thus, in the worst case, all rewritings are enumerated before a minimal one is returned. A simple improvement is **Increasing-Subset-Enumeration**, or **ISE**, which builds $\mathcal{W}$ subsets from the smallest to the largest. Using a proper trie structure for R, one can efficiently check if a subset of the views used in a rewriting has already lead to another rewriting, and if so, discard the larger one.

Algorithm ISE repeats a lot of work. For example, let v_7 be the view b_{val} and v_8 be the view $b_{c_{id}}$. They cannot be joined on b, and any $\mathcal{W}$ subset including them both will not lead to a rewriting. However,

Algorithm ISE will try such subsets. Similarly, if v_9 and v_{10} can be joined, then this partial result could be stored to be re-used in several larger rewritings.

Based in this intuition, we devise a bottom-up, **Dynamic Programming Rewriting** algorithm (or **DPR**, in short). It attempts to build larger and larger partial rewritings, by combining smaller ones. The initial set of rewritings is made of the pairs of ($\mathcal{W}$ view, embedding in the query). Then, DPR combines an existing rewriting, and a rewriting made of only one view, akin to building left-deep plans during optimization. However, unlike an optimizer, DPR only explores one ordering per sets of views, exactly to avoid doing the optimizer's work. To combine two partial rewritings, namely e_1 over the set of views V_1 and set of embeddings Φ_1, and e_2 similarly based on V_2 and Φ_2, DPR invokes Algorithm 1 on $V_1 \cup V_2$ and $\Phi_1 \cup \Phi_2$. Coming back to the above examples, DPR will observe that v_7 and v_8 cannot be combined, and not attempt a rewriting combination if $\{v_7, v_8\}$ is a subset of $V_1 \cup V_2$. The partial rewriting joining v_9 and v_{10}, returned by Algorithm 1, will be used to build larger rewritings using one extra view. This give DPR a significant reduction of work over ISE.

Algorithm DPR will identify a rewriting of k views only after having tried all rewritings using up to $k-1$ views, which may take too long.

To alleviate this, we propose the **Depth-First Rewriting** algorithm (or **DFR**, in short). Like DPR, it is bottom-up, and it builds only minimal, left-deep rewritings. However, instead of exploring all combinations of increasingly many views, DFR is a greedy algorithm. At any moment, it picks the partial rewriting *covering the most query nodes* found so far, and joins it with a 1-view partial rewriting. This leads DFR to frequently finding a first rewriting very fast. In exchange, when DFR tries a set V of views, its subsets may have not been previously explored. For instance, it may explore $\{v_7, v_8\}$ after $\{v_7, v_8, v_{12}\}$ and after $\{v_7, v_8, v_{13}\}$, thus discover the incompatibility of v_7 with v_8 several times.

ISE, **DPR** and **DFR** are correct and complete; they produce the minimal rewritings of q given $\mathcal{V}$. ISE and DPR produce the rewritings having the fewest number of views possible, before the others. ISE and to a lesser extent DFR may repeat some work. ISE and DPR produce rewritings towards the end of the search, whereas DFR may produce some very early on.

4.3 Evaluating a rewriting

A logical rewriting plan must be optimized by standard algebraic transformations, e.g. transforming the $\sigma(\times)$ into a join tree, pushing σ and π etc., and then trasformed into a physical plan. In ViP2P, this plan is typically distributed over the peers in the DHT. The execution engine includes standard implementations for $scan$, σ, π, hash joins, binary structural joins [2] and a holistic twic structural join [8]. In the view definition index, we annotate the view tree pattern with its cardinality (known at the view peer), allowing the optimizer to decide about join orders. The optimizer applies heuristics to reduce, first, inter-site transfers, and second, the number of sort operations.

5 P2P view management

We have so far explained how to exploit views for query rewriting. We now consider how views are materialized (Section 5.1), and identified in order to rewrite a query (Section 5.2) in the DHT network. Both operations require some *view definition indexing* in the DHT. We stress that we do not index view extent (tuples), but only the pattern defining the views.

We start by introducing a useful term: if d is a document and v is a view such that $v(d) \neq \emptyset$, we say d *affects* v.

5.1 View materialization

Assume peer p decides to establish a view v. Then, when a peer p_d publishes a document d affecting v, p_d needs to find out that v exists. To that effect, view definitions are *indexed for document-driven lookup* as follows. For any label (node name or word) appearing in the definition of the views $v_1, v_2, \ldots, v_k$, the DHT will contain a pair where the key is the label, and the value is the set of view URLs $v_1, v_2, \ldots, v_k$.

When a peer p_d publishes a document d, p_d performs a lookup with all d labels (node names or words) to find a superset S_a of the views that d might affect. Then, p_d evaluates $v(d)$ for each $v \in S_a$. We implemented this step based on a SAX traversal, with time complexity in $\Theta(|d| \times |v|)$. In practice, large fragments of d are typically not interesting for a given view v, thus computing $v(d)$ tends to spend some time traversing useless parts of d. To share this cost, we group view definitions in batches of some size n (we set $n = 10$) and evaluate all the views of a batch in a single d traversal. Thus, d fragments useless to all the views in a batch, are parsed only once per batch.

Finally, p_d sends, for each view v, the tuple set $v(d)$ (if it is not empty) to the peer p_v publishing v. Recall from Section 2 that element IDs include document URIs, which may get rather lengthy. To speed up transfers, tuples are encoded so that the URI of d is sent only once for the tuple set $v(d)$.

We have so far considered that v is published before the documents which affect it. The opposite may also happen, i.e. when v is published, a document d affecting v may already exist, and $v(d)$ needs to be added to v's extent. To that effect, we require the publisher p_d of a document d to periodically look up the set of views potentially affected by d, and send $v(d)$ to those views as described above. Thus, v will be up to date (reflecting all network documents that affect it) after the periodical check and subsequent actions have been performed by all document publishing peers.

We end the section by considering view maintenance in the face of document deletion or change. When documents are deleted from the system, a similar view lookup is performed, and the peers holding the views are notified to remove the respective data. We model document changes as deletions followed by insertions.

5.2 Identifying views for rewriting

A second form of view definition indexing is performed in order to find views that may be helpful for rewriting a given query. In this context, a given algorithm for extracting (key, value) pairs out of a view definition is termed a *view indexing strategy*. For each such strategy, a *view lookup* method is needed, in order to identify, given a query q, (a superset of) the views which could be used to rewrite q. Many strategies can be devised. We present four that we have implemented, together with the space complexity of the view indexing strategy, and the number of lookups required by the view lookup method. We also briefly show that these strategies are *complete*, i.e. they retrieve at least all the views that could be embedded in q and, thus, lead to q rewritings.

Label indexing (LI): index v by each v node label (either some element or attribute name, or word). The number of (key, value) pairs thus obtained is in $O(|v|)$.
View lookup for LI: look up by all node labels of q. The number of lookups is $\Theta(|q|)$.
LI completeness is quite straightforward (details omitted).
The LI strategy coincides with the view definition indexing for document-driven lookup (described in the previous section). An interesting variant can furthermore be devised:
Return label indexing (RLI): we index v by the labels of all v nodes which project some attributes (at most $|v|$).
View lookup for RLI, interestingly, is the same as for LI. The labels on which LI indexes v, and RLI doesn't, are those of v nodes without attributes. On such nodes, no join can be applied on v (due to the lack of id), and no navigation (due to the lack of $cont$). Moreover, such nodes obviously do not provide attributes corresponding to those returned by the query. Therefore, one does not need to advertise v based on their labels.

The drawback of LI and RLI is their lack of precision. For instance, a view $a_{id}(c_{id})$ will be retrieved for all queries involving the terms a, although it is useless for all queries not containing c. A more precise strategy is the following.

Leaf path indexing (LPI): let $LP(v)$ be the set of all the distinct root-to-leaf label paths of v. *In this context, a path is just a sequence of the node names, it does not include the edges.* Index v using each element of $LP(v)$ as key. The number of (key, value) pairs thus obtained is in $\Theta(|LP(v)|)$.
View lookup for LPI: let $LP(q)$ be the set of all the distinct root-to-leaf label paths of q. Let $SP(q)$ be the set of all non-empty sub-paths of some path from $LP(q)$, i.e., each path from $SP(q)$ is obtained by erasing some labels from a path in $LP(q)$. Use each element in $SP(q)$ as lookup key.

As an example, let $v = a_{id}(b_{id}, c_{id})$, then v will be indexed by the keys $a.b$ and $a.c$. Let q be the query $a(f(b_{id}, c_{id}))$. With LPI, the view lookups will be on a, $a.f$, $a.b$, $a.c$, f, $f.b$, $f.c$, b, and c. Thus, v will (correctly) be identified as potentially useful to rewrite q. Indeed, if a view $v' = f_{id}$ exists, then $q = \sigma_{a \prec f \wedge f \prec b \wedge f \prec c}(v \times v')$.

Let $h(q)$ be the height of q and $l(q)$ be the number of leaves in q. The number of lookups is bound by $\Sigma_{p \in LP(q)} 2^{|p|} \leq l(q) \times 2^{h(q)}$.
LPI completeness: observe that if a view v can be embedded in the query q, then $LP(v) \subseteq SP(q)$.

The last strategy we consider is:
Return Path Indexing (RPI): let $RP(v)$ be the set of all rooted paths in v which end in a node that returns some attribute. Index v using each element of $LP(v)$ as key. The number of (key,value) pairs is also in $\Theta(|RP(v)|)$.
View lookup for RPI coincides exactly with the lookup for LPI. RPI completeness is shown similarly to RLI.

6 Performance evaluation

In this section, we present a set of experiments we made to estimate the performance of various aspects of our architecture. Section 6.1 briefly describes our platform. Section 6.2 presents the experimental setup for the next two sections: Section 6.3 considers view materialization, while Section 6.4 studies query processing. Section 6.5 studies view indexing and lookup techniques, whereas Section 6.6 focuses on query rewriting on one peer. Section 6.7 concludes our study.

6.1 System implementation and configuration

We have fully implemented the platform described so far, using Java 6. Berkeley DB (version 3.3.75, available from www.oracle.com) and FreePastry (version 2.1, available from freepastry.org) are used for storing view data and indexing view definitions respectively. For the implementation of the nav operator, patterns are translated to XQueries, and executed by the Saxon XQuery processor (version Saxon-B 9.1, available from saxon.sourceforge.net). The nav operator is always placed on the same peer as its input, thus it is evaluated locally.

We have made some optimizations to speed up inter-peer data transfers. More precisely, when sending a stream of tuples, potentially including many document URIs in node IDs, we encode the URIs on the fly in compact integers, and send the dictionary with the tuples, so that they can be decompressed on the other side.

In our experiments, unless otherwise specified, we have deployed **1000 ViP2P peers on 250 machines** on the Grid5K research network (https://www.grid5000.fr). Each machine hosts 4 peers. The machines are distributed across 9 big French academic centers. They have between 2 GB and 4 GB of memory; most of them are multi-cores. All run 64 bits Debian Linux 2.6.18. *Due to Grid5K restrictions, we could not reserve the same sets of machines for all experiments.* We ran most experiments three times and averaged

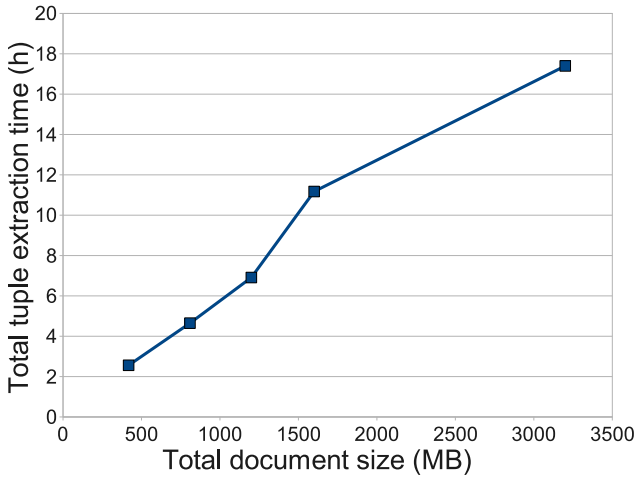

Figure 5: Total tuple extraction time.

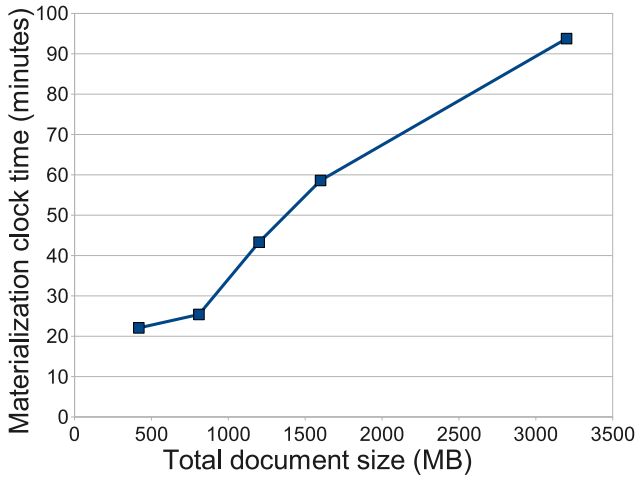

Figure 6: Observed view materialization latency.

the times; the difference between 2 runs was up to 20% of the values, but the general tendencies were stable.

6.2 Setup for view building and query processing

The peers publish a total of 2000 XMark benchmark documents [15] of equal size; the total size of the network documents varies across successive runs, from 400 MB to 3.2 GB. The peers also publish 500 views of up to 7 nodes. 70 views are affected by all documents; the others use XMark node names but have no results on XMark documents. The documents and views are split uniformly over the network. The views are indexed using LI. All 500 views are retrieved for all 2000 documents by the document-driven lookup method described in Section 5.1.

Once views are indexed, a designated *coordinating peer* sends to all others a *start* signal. Then, in parallel, the peers look up views, extract data, send and receive tuples, and store them in their local BerkeleyDB databases. After all its tuples are stored, each peer sends a *done* signal to the coordinating peer. Of course, this synchronization is just for the experiment, and is not needed otherwise.

6.3 View building

Figure 5 shows the total time needed to evaluate 500 views on the 2000 documents. Extraction takes place at the documents' sites. The times are summed up for all the peers; in reality, extraction takes place in parallel. As expected from the description in Section 5.1, extraction time grows linearly with the total document size.

Figure 6 shows the time measured at the coordinating peer, between its *start* signal and the last of the 1000 *end* signals. It can be seen as the time to load the network with our documents and views, at the fastest possible pace. The time grows linearly in the data size, as was to be hoped.

Data transfers for view materialization increased linearly in the size of the documents. For the 3.2 GB of published data, we transferred 468 MB of data for view materialization, after URI compression.

6.4 Query evaluation

Once the views are loaded, we ask the query:
$site_{id}(regions_{id}(africa_{id}(item_{id})), catgraph_{id}(edge_{id}))$.

Query rewriting & optimization at the query peer take, respectively, 30 ms and 100 ms. The smallest rewriting uses two views on two machines, different from one where the query is asked.

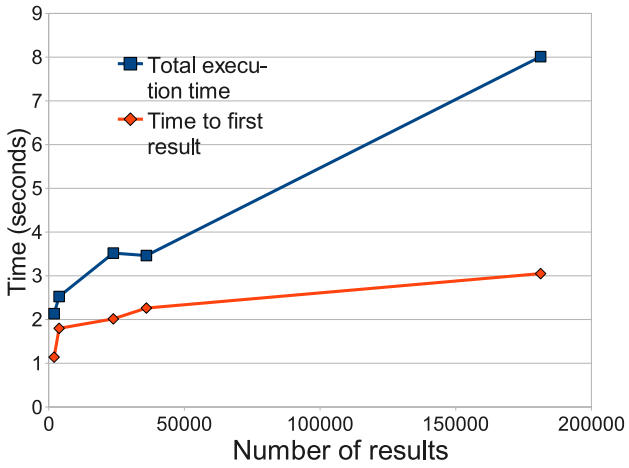

Figure 7: Execution time for increasing data size.

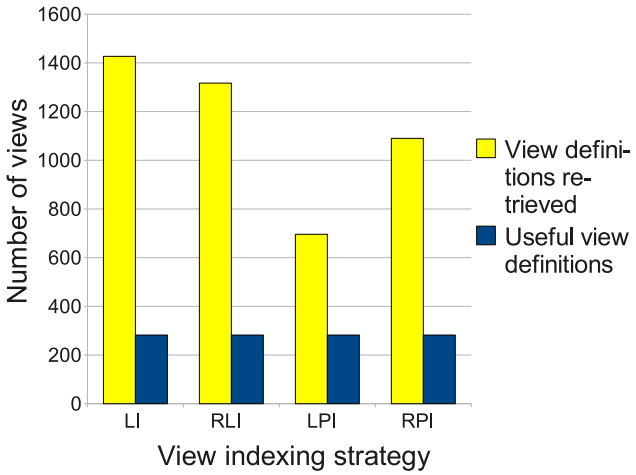

Figure 8: View definition retrieval.

Query execution Figure 7 shows that as expected, query execution time scales up with the size of the data set.

Data transfers for query processing also grew linearly with the total document size, up to 12.57 MB for processing our query on the 3.2 GB document set.

The benefits of VIP2P views can be appreciated on the following simple example. We use a data set of 750 XMark [15] documents having the total size of 20 MB. We use three different view sets to rewrite the query $site(item(description_{cont}))$:

- $\mathcal{V}_1$ contains the view $site_{cont}$. This corresponds to storing the full documents in one single view; we use it to have a glimpse of the interest of *document-level granularity* indices. Indeed, a system such as [11] would identify all the corresponding documents and then evaluate the query on the fly on those documents. We proceed quite in the same way, by our rewriting $nav_{site_{cont},item(description_{cont})}(v_1)$.
- $\mathcal{V}_2$ contains three views: $site_{id}$, $item_{id}$ and $description_{id,cont}$. This corresponds to the node-granularity indexing used in [1], but unlike [1], we also time the transfer of the XML results to the query peer.
- $\mathcal{V}_3$ contains one view which is exactly q.

This experiment was made with 2 peers in a 10 GB LAN, to minimize data transfer impact. The view lookup and rewriting times are negligible; the execution times are: 8.8 seconds for $\mathcal{V}_1$; 2.1 seconds for $\mathcal{V}_2$; and 1 second for $\mathcal{V}_3$. As expected, having a view exactly matching the query is best. This exemplifies the query speed-up that can be obtained using views, if we pay the cost of building them.

6.5 View indexing and lookup strategies

In this section, we compare the view indexing and lookup strategies LI, RLI, LPI and RPI described in Section 5. We consider a synthetic query q of 30 nodes labeled $a_1, \ldots, a_{30}$. Each node of q has between 0 and 2 children, and q's height is 5. From q, we create three variants:

- q' has the same labels as q, but totally disagrees with q on the structure (whenever a_i is an ancestor of a_j in q, this does not hold in q')
- q'' coincides with q for half of the query (one child of the root), while the other half conserves the corresponding q labels but totally changes structure (as q' does)
- q''' has the same structure as q, half of it has the same labels $a_1, \ldots, a_{15}$, while the other half uses a different set of tags $b_1, \ldots, b_{15}$ (instead of $a_{16}, \ldots, a_{30}$).

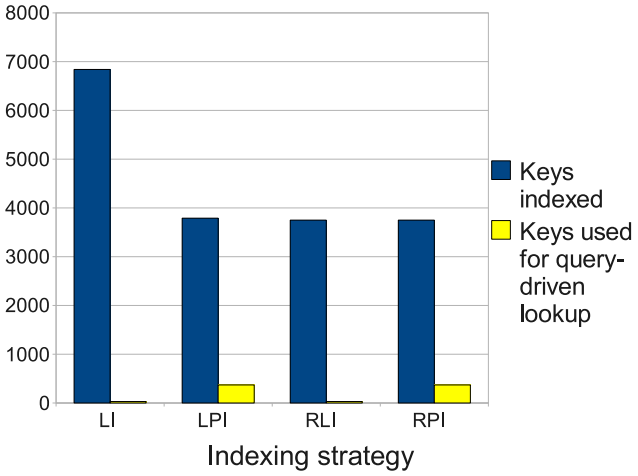

Figure 9: Index entries and lookups generated for the views.

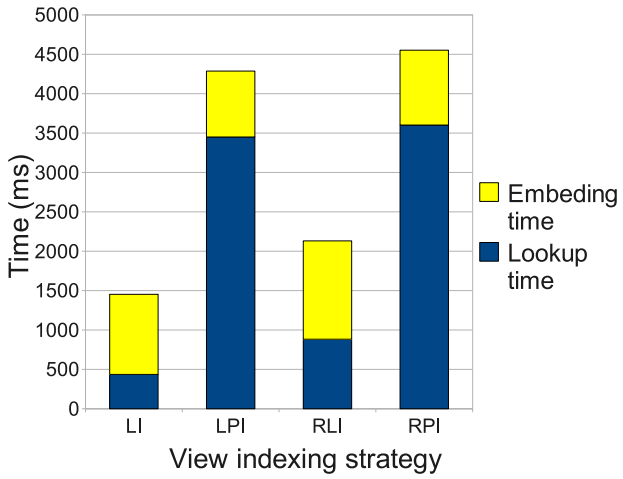

Figure 10: Identifying useful views.

From each of q, q', q'' and q''' we automatically generate 360 views of 2 to 5 nodes, for a total of 1440 views. The views can all be embedded into the respective queries, i.e. those generated from q can be embedded in q, those generated from q' can be embedded in q' and so on. We, thus, obtain a mix of views ressembling the query to various degrees. To this randomly-generated view set, we added 3 hand-picked views to ensure that one query rewriting exists.

We have indexed the resulting 1443 views in our network, following the LI, RLI, LPI and RPI strategies described in Section 5. We then performed the four corresponding lookups.

Figure 8 shows how many views have been retrieved for each strategy, compared with the number of useful views (those that are found to be embeddable in q, in our example, those generated from q, and possibly some generated from q'' and q'''). We see that the path indexing-lookup strategies (LPI and RPI) are more precise than label based ones (LI and RLI). Moreover, LPI is the most precise. This is because LPI uses longer paths as keys, thus, it describes views more precisely, eliminating some false positives.

Figure 9 presents the number of (key, value) pairs added to the index by each view indexing strategy, and the number of lookups needed by each strategy for the query we considered. As expected, LI leads to most index pairs. With respect to query-driven lookup, LI and RLI lead to 30 lookups, much less than LPI and RPI lead to 370 lookups.

Figure 10 shows the time to obtain the initial set of views. The Figure distinguishes the time to perform in parallel all the lookup calls on Pastry, and the time to test if each view is useful by embedding it into q. The Figure shows that the simple LI strategy is the best. Indeed, even though Pastry lookups are asynchronous, issuing many lookups from the same peer comes with a penalty, thus, LPI and RPI, which needed 370 lookups, are significantly slower. LI makes up for its low precision by requiring few lookups.

We mention that rewriting the query based on the relevant views (282 in this example) takes around 6 seconds, whereas finding the first solution takes around 0.5 seconds. Comparing this with the times in Figure 10, one notices that view definition look-up is quite short, which validates the feasibility of retrieving view definitions at query rewrite time. One may also consider *locally caching* view definitions, to completely avoid look-ups. The view pruning time could further be reduced as we explain in Section 7.

6.6 Query rewriting

We use queries of 5, 9, 13 and 17 nodes, respectively. Each query is a balanced binary tree where all internal nodes have two children. All nodes have different labels; the root has id, the other nodes have no attributes. This experiment ran on a MacBookPro on the Darwin 9.6.0 kernel, and having a 2.5 GHz Intel Core 2 Duo processor.

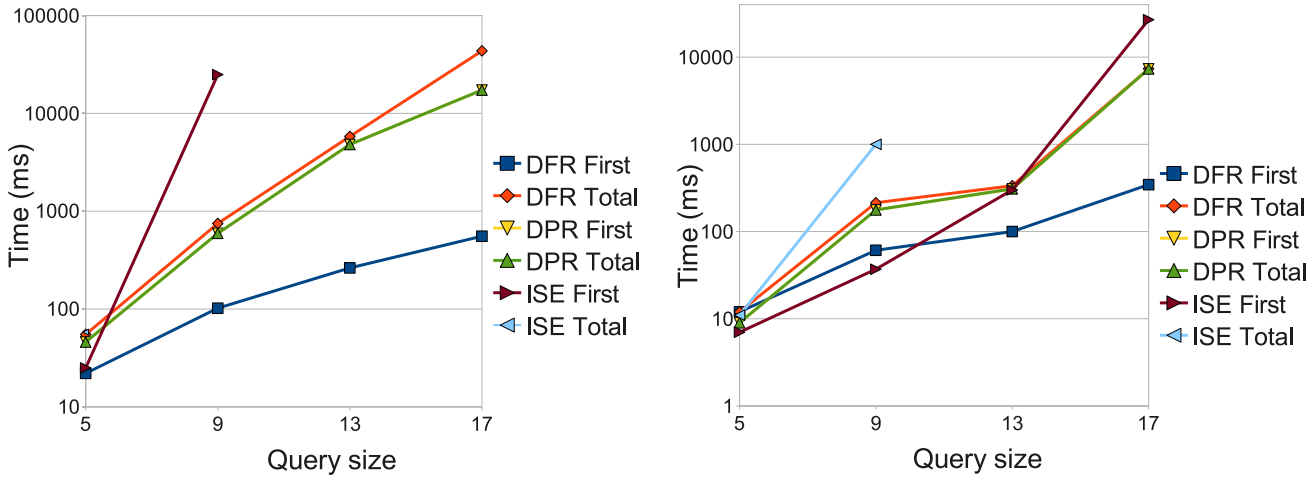

Figure 11: Query rewriting times.

First, for each query, we make a set of $|q|$ 1-node views, one per query label, each having an id. This is a very hard case for our bottom-up algorithms, as almost any subset of views can be joined. The expected complexity here is $O(|q|^{|q|+2})$.

Second, we devise for each query another set of approx. $|q|/2+1$ views. One of these views copies the top 2 levels of the query nodes; the remainder ones are small subtrees of 1-3 nodes, made of the lowest levels in the query trees. In these sets of views, only about half of the nodes have ids (we took care that rewritings still exist). The complexity her is in $O((|q|/2)^{|q|+2})$. Reducing id presence also reduces the join opportunities and thus, simplifies the problem. In both cases, all the views can be embedded in the query.

Figure 11 shows, for the first family of view sets (top) and the second family (bottom) the total time, and the time to the first rewriting, taken by ISE, DPR and DFR. The missing points are times longer than 2 minutes. The highest times are 43 seconds for DFR-Total (at the top) and 27 seconds for ISE-First (at the bottom). Recall that the complexity of the problems we study is in $O(|\mathcal{V}|^{15})$. Figure 11 shows, first, that ISE does not scale; the total time is very large even for $|q|=9$ in the upper graph. Second, as expected, for DPR the total time and the time to the first solutions almost coincide. Third, DFR reaches a first solution much faster than the others; we checked and these first rewritings were also of the minimal size (although this cannot be guaranteed in general). Fourth, DFR total time is indeed longer than DPR's, due to the fact that DFR may repeat some work since it does not explore subsets the increasing order of their sizes.

Finally, we consider again the 17-nodes query and the 9 views used in lower part of Figure 11. We add a view of 1 node, with the label of the query root, and having $cont$. The query root has 2 children, thus, as explained in Section 3.2, expansion transforms this view into 4 views (and not 2^{17}). Thus, rewriting proceeds with 13 views. Now, the smallest rewriting uses just the fully expanded view; DFR, DPR and ISE all find it in less than 100 ms. The total times are respectively 5.8 seconds, 4.3 seconds, and more than 2 minutes.

6.7 Conclusions of the experiments

Our experiments show that the VIP2P approach for view materialization and query processing scales up linearly in the data size, on a network of 1000 peers. With respect to the rewriting problem, when queries are complex and/or there are many views, DFR tuned to stop after the first rewriting gives reasonable performance. Rewriting time is also strongly correlated to the number of ids in the views, since they enable joins.

View indexing and lookup are relatively fast, which validates the feasibility of exploiting views distributed over the peer network. Among the view indexing strategies we compared, LI cuts the most interesting compromise between precision, number of entries in the DHT index, and number of lookups

needed for a given query to rewrite.

On one simple example, we have demonstrated the potential for performance improvement provided by VIP2P views, over DHT indexes either at document granularity level, or at node level. This demonstrates that there exists a large in-between space, where views closely suited to application needs can provide significant performance benefits.

7 Related works

Our work is related to view-based XML query rewriting using, and to distributed XML data management.

Tree pattern query rewriting Rewriting an XPath query based on an XPath view has been studied in [6, 21]. More recent works have considered rewriting XPath queries using multiple views. View intersection is used to build rewritings in [9], and the DAGs we use in Section 4.1 recall their study, since ID equality join is akin to intersection. Our rewriting problem is complicated by the fact that our views have multiple attributes at various places in the view. Thus, we need joins, and we need to take into account how many times a tuple is multiplied by each extra join (as in the discussion around expansion and Figure 2). Also, we assume structural ids, which enable e.g. rewriting $a(b_{cont})$ out of a_{id} and $b_{id,cont}$, which [9] does not handle. The recent work of [18] takes structural ids a step further. They use XPath views (including wildcard nodes labeled $*$) where the return node always has $cont$ *and* a powerful structural id, encapsulating the ids and labels of all its ancestors, up to the root. Thus, unlike us and [9], they may rewrite $a(b_{cont})$ using $b_{id,cont}$, simply by checking the $b.id$ for an a-labeled ancestor. We chose not to adopt such ids since they are rather lengthy, and their encoding relies on an NFA [12]. In our context, querying many documents, each of which would need an NFA, would significantly increase node id size, and thus, potentially data transfers. Rewriting is reduced in [18] to finding covers of the query leaves. Our rewritings need to cover the whole query, but we have proved in Section 4.1 a $|q|$ bound on the rewriting size, and polynomial complexity for the rewriting. In contrast, in [18] the rewriting size bound is $|\mathcal{V}|$ and the complexity is exponential in the number of query leaves. View embedding in the query is very expensive in the presence of $*$, thus [18] prunes views by building a view automaton at a cost of $\Sigma_{v\in\mathcal{V}}(|v|)$, and then running q through the automaton. We could also apply this; it would reduce our pruning cost (e.g., the embedding time in Figure 10) by a factor of $|q|-1$.

Rewriting rich patterns with multiple attributes is studied in [4], under Dataguide constraints which strongly impact the algorithm, and without considering distribution. XQuery rewriting based on XQuery views is studied in [14], which establishes polynomial complexity for the XPath case.

From XQuery to tree patterns More generally, tree pattern views with multiple attributes allow answering more queries than XPath views (the presence and properties of node IDs also impacts the queries which may be answered, as shown above). For instance, $article(abstract_{id,cont}, author_{id,val})$ allows answering both $article(abstract_{cont})$ and $article(author_{val})$ (use π and duplicate elimination on some ids). Rich tree patterns, including optional and nested edges, come very close to capturing an XQuery dialect of nested FLWR (for-where-return) expressions [5]. In particular, the mandatory part of a nested FLWR query is found in the for-where clauses of the outermost block, and is captured by a conjunctive pattern, as considered in this work.

[22] describes efficient XQuery evaluation techniques, for queries over documents whose URIs are known (without using views or a DHT). The benefits of such techniques are orthogonal - and could be cummulated with - those of using pre-computed view results, as we advocate in this work.

Distributed XML processing Closest to our work are techniques for indexing and querying XML in DHT networks [11, 7, 17, 1]. Each of these works uses a specific single XML indexing strategy, whereas we propose more flexible views, which can be better tailored to the query needs. View definitions are indexed on a DHT in [16], but they consider RDF data and rewritings based on only one view.

A previous version of this work has been informally presented in a workshop (not in the proceedings) [13]. The complexity analysis and experiments presented here are new.

8 Conclusion

The efficient management of large XML corpora in structured P2P networks requires the ability to deploy data access support structures, which can be tuned to closely fit application needs. We have presented the VIP2P approach for building and maintaining structured materialized views, and processing peer queries based on the existing views in the DHT network. Using DHT views adds the cost of a view definition lookup, but pre-computed views can strongly reduce query evaluation times. We have characterized the complexity of rewriting conjunctive tree pattern queries with attributes, using materialized views, and we have compared several algorithms for view-based query rewriting; DFR seems to be the most useful. We studied several view indexing strategies and associated complete view lookup methods. The LPI method seems best, due to its low cost both in DHT messages involved in indexing and lookup, and to its good precision.

Many avenues for future work exist. To efficiently handle very large views, we could employ horizontal view fragmentation, which would parallelize query execution, as was done for the DHT index in [1]. Collaborative view recommendation is a next step; algorithms for the centralized case start to appear [19]. Also, we are currently extending the view pattern language presented here with value joins, to handle queries over XML documents with RDF annotations.

Acknowledgments

We thank Alin Tilea and Silviu Julean for their help in developing and testing ViP2P. This work has been partially funded by *Agence Nationale de la Recherche*, decision ANR-08-DEFIS-004.

References

[1] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE*, 2008.

[2] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[3] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4), 2002.

[4] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, 2007.

[5] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou, and R. Vijay. Algebra-based identification of tree patterns in XQuery. In *FQAS*, 2006.

[6] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[7] A. Bonifati and A. Cuzzocrea. Storing and retrieving XPath fragments in structured P2P networks. *Data Knowl. Eng.*, 59(2), 2006.

[8] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[9] B. Cautis, A. Deutsch, and N. Onose. XPath rewriting using multiple views: Achieving completeness and efficiency. In *WebDB*, 2008.

[10] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured P2P overlays. In *Proc. of IPTPS*, 2003.

[11] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB*, 2003.

[12] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen. From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In *VLDB*, 2005.

[13] I. Manolescu and S. Zoupanos. XML materialized views in P2P. DataX workshop (not in the proceedings), 2009.

[14] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD*, 2006.

[15] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[16] L. Sidirourgos, G. Kokkinidis, T. Dalamagas, V. Christophides, and T. K. Sellis. Indexing views to route queries in a PDMS. *Distributed and Parallel Databases*, 23(1), 2008.

[17] G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient processing of XPath queries with structured overlay networks. In *CoopIS*, 2005.

[18] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong. Multiple materialized view selection for XPath query rewriting. In *ICDE*, 2008.

[19] N. Tang, J. X. Yu, H. Tang, M. T. Özsu, and P. A. Boncz. Materialized view selection in xml databases. In *DASFAA*, 2009.

[20] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD Conference*, 2002.

[21] W. Xu and M. Ozsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.

[22] Y. Zhang, N. Tang, and P. A. Boncz. Efficient distribution of full-fledged XQuery. In *ICDE*, 2009.